LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 230 – Abgabe zu Aufgabe A204
Sommersemester 2022

Vladislav Dzhorov Eslam Nasrallah Yiyang Xie

1 Einleitung

Das Ziel dieser Projektaufgabe ist ein Programm zum Dekomprimieren von BMP-Grafiken zu erstellen. Dafür muss man im Zuge der Projektaufgabe theoretisches Wissen aus der Mathematikum Anwendungszusammenhang verwenden und einen Algorithmus in C implementieren.

Die Ursprünge der Datenkompression beruhen auf der Informationstheorie. [2]
Im Wesentlichen bedeutet "Kompression und Dekompression" die Wahrscheinlichkeitsverteilung des Inhalts einer Datei zu ermitteln. Bei der Kompression wird die Redundanz beseitigt, bei der Dekompression wird die Redundanz wiederhergestellt. Es ist das Äquivalent dazu, denselben Inhalt in einer anderen Form auszudrücken.

1.1 Analyse der Projektaufgaben

Die Aufgabe des praktischen Teils des Projekts besteht darin, eine komprimierte Bitmap einzulesen und sie mit Hilfe des RLE-Algorithmus in eine unkomprimierte Bitmap umzuwandeln.

Zu Beginn des Projekts wird ein Rahmenprogramm erstellt. Man soll darin C-I/O-Operationen implementieren, mit denen man eine BMP-Datei einlesen und einen Pointer auf diese Bilddaten und die zugehörigen Metadaten (z.B. Höhe und Breite) an die C-Funktion übergeben können. Anschließend sollte man eine neue BMP-Datei erstellen und die berechneten Werte nach dem Aufruf in diese schreiben. Das Ergebnis sollte mit dem üblichen Bildbetrachter lesbar sein.

Die Hauptfunktion ist:

$void$ bmp_rld($const\ uint8\_t^*$ rle_data, $size\_t$ len, $size\_t$ width, $size\_t$ height, $uint8\_t^*$ img)

- $const\ uint8\_t^*\ rle\_data$: Dies ist das mit RLE komprimierte BMP-Bild, das der Benutzer dem Programm zur Verfügung stellt.
- $size\_t\ len$: Zusätzlich wird der Benutzer die Länge von rle_data angeben. Dies wirkt sich auf die Länge des Lesevorgangs und die Anzahl der Iterationen des Programms aus.

- $size\_t\ width,\ height$: Die Breite und Höhe des Bildes, die sich auf die Größe und Auflösung des Bildes beziehen. Dies wirkt sich auf die Zuweisung von Speicherplatz durch das Programm aus.
- $uint8\_t^*\ img$: Schließlich die Ausgabe des Programms. Abhängig von den oben genannten Parametern gibt das Programm ein dekomprimiertes BMP-Bild aus, das an dem Ort gespeichert wird, auf den dieser Zeiger zeigt.

2 Lösungsansatz

Im Folgenden werden die Konzepte und konkrete Implementierung des Bilddekompression Algorithmus tiefer betrachtet.

Der Benutzer kann RLE-Daten in einem komprimierten BMP bereitstellen, wie es die Aufgabe erfordert. Die Größe der Daten beträgt 1Byte/8bit pro Einheit, weil das Programm nur das 8bpp-BMP-Format abdeckt. Wenn die Daten nicht in diesem erforderlichen Format vorliegen, kann das Programm die Datei nicht dekomprimieren.

2.1 Konzepte

Nach der Erläuterung der Projektaufgaben ist es wahrscheinlich, dass Personen, die nicht an dem Projekt teilgenommen haben, über einige Begriffe verwirrt sein werden. Daher ist es wichtig, die grundlegenden Konzepte zu verstehen, bevor wir den Lösungsansatz unserer Gruppe vorstellen.

2.1.1 Bitmap

Bitmap ist eine Form der Beschreibung eines Bildes in Form von computerlesbaren Daten. Windows Bitmap (BMP) ist das Standard Bilddateiformat für das Windows Betriebssystem und wird von einer Vielzahl von Windows Anwendungen unterstützt. Die Standard-Dateiendung für BMP-Bitmap-Dateien ist $.bmp$. Es besteht aus sogenannten Pixeln, denen jeweils eine Farbe zugeordnet ist. Diese Pixel können auf unterschiedliche Weise angeordnet werden, um verschiedene Muster zu bilden. [4]

2.1.2 Lauflängenkodierung

Die Lauflängenkodierung bzw. RLE ist ein Algorithmus, der in Windows zur Kompression von Bilddateien verwendet wird. Windows unterstützt Formate zum Komprimieren von Bitmaps, die ihre Farben mit 8 oder 4 Bit pro Pixel definieren. [1]

Gemäß den Anforderungen dieses Praktikums muss das Programm explizit nur das 8bpp-BMP-Format abdecken!

Die Grundidee ist, dass benachbarte Pixel mit derselben Farbe in einer Scanzeile durch

zwei Bytes dargestellt werden. [9] Das erste Byte ist die Anzahl der Wiederholungen des Pixels, das Zweite ist der Wert oder die Farbe des Pixels, z.B.

RRRRRRRRGGBBBBBB ——> 8R2G6B

Nach der RLE-Kompression ist die Länge der Zeichenfolge signifikant kürzer als vorher.

2.2 Allgemeiner Algorithmus

An dieser Stelle kann man davon ausgehen, dass man die Projektaufgabe, die Hauptfunktion, die Parameter und die Grundkonzepte verstanden hat.

Der Algorithmus zur Bilddekompression funktioniert wie folgt. [7]

- Alle 2 Bytes enthalten Informationen über eine Sequenz mit einer einzigen Farbe.
  - Das erste Byte ist die Anzahl der Wiederholungen des Pixels.
  - Das zweite Byte ist der Wert oder die Farbe des Pixels.
- Wenn das erste Byte Null ist, wird das zweite Byte betrachtet.
  - Wenn es ebenfalls 0 ist, bedeutet dies das Ende der Zeile.
  - Wenn es eine 1 ist, bedeutet dies das Ende der Datei.
  - Wenn es eine 2 ist, bedeutet dies das Delta.
  - Die 2 Bytes nach dem Escape enthalten vorzeichenlose Werte, die den Versatz nach rechts und nach oben des nächsten Pixels von der aktuellen Position angeben, z.B.
    Die Bedeutung von [00 02 05 01] ist, dass der aktuelle Zeiger 5 Byte nach rechts und dann eine Zeile nach unten geschoben wird, d.h. ein Teil der Pixel wird übersprungen.
- Zum Schluss werden die dekodierten Daten im $uint8\_t^* \; img$ gespeichert.

Hinweis: RLE-Daten verwenden eine spezielle Struktur, um mehrere, sich nicht wiederholende Datenpixel zu speichern, z.B.

- [00 03 45 56 67 00] dekomprimiert zu [45 56 67].
- [00 04 39 56 67 12] dekomprimiert zu [39 56 67 12].

d.h. das erste Byte ist die Kennung “00“. Das zweite Byte hat einen Wert größer als 2, der die Länge der sich nicht wiederholenden Pixeldaten angibt (Zeile 5) [7]. Jedes nachfolgende Byte sind die spezifischen Pixeldaten.

Wenn das zweite Byte eine gerade Zahl ist, ist die Struktur wie oben beschrieben.
Wenn das zweite Byte eine ungerade Zahl ist, wird am Ende dieser Dateneinheit eine zusätzliche “00“ angefügt.
Das Ignorieren von Farben, die sich nicht wiederholen, führt nach der Dekompression zu einem erheblichen Datenverlust.

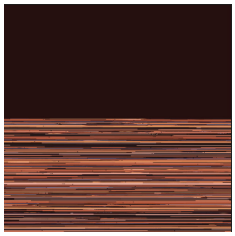


Abbildung 1: Fehlerbeispiel

Abbildung 2: Korrekte Ausgabe

(Nur repräsentative Beispiele, weitere Tests finden sich in der praktischen Abteilung.)

2.3 Implementierung in C

Ausgehend von der Struktur der RLE-Daten kann man sich intuitiv vorstellen, eine Schleife zum Lesen und Verarbeiten der Daten zu verwenden. Die Anzahl der Iterationen wird durch die Länge der RLE-Daten begrenzt, d.h. durch den Parameter $size\_t\ len$. Die Schleife liest die Daten der Größe 2 Bytes und schreibt dann das Ergebnis in einer verschachtelten Schleife an die Stelle, auf die $uint8\_t^*\ img$ zeigt.

Der abgekürzte Code lautet wie folgt, der vollständige Code ist in $bmp\_rld\_c\_basic$ in der Datei $main.c$.

```
1  void bmp_rld_c_basic(const uint8_t *rle_data, size_t len, size_t width,
       size_t height, uint8_t *img) {
2      ...      // initialization
3
4      for (size_t i = 0; i < len;) {
5          if (rle_data[i] == 0 && rle_data[i + 1] > 2) {
6              ...      // non-repeating colors
7          } else if (rle_data[i] == 0 && rle_data[i + 1] == 0) {
8              ...      // end of line
9          } else if (rle_data[i] == 0 && rle_data[i + 1] == 1) {
10             ...      // end of bitmap
11         } else if (rle_data[i] == 0 && rle_data[i + 1] == 2) {
12             ...      // delta marker
13         } else {
14             ...      // normal decoding with loop
15             for (; j < limit; j++) {
16                 ...      //copy pixels byte by byte
17             }
18         }
19     }
20 }
```

Der Vorteil dieser Implementierung in C ist, dass der Code logisch verständlich, mit guter Lesbarkeit ist und schnell implementiert werden kann.

2.4 Implementierung in Assembler x86-64

Diese Implementierung folgt der obigen C-Implementierung. Das Rahmenprogramm ist in C geschrieben und die Bilderdekompression-Funktion ist in Assembler implementiert.

Der vollständige Code befindet sich in der Datei $bmp.S$.

2.5 Implementierung in Assembler x86-64 mit SIMD

Bei der SIMD-Implementierung werden 16 Pixels gleichzeitig verarbeitet. Das Verfahren ähnelt der SISD-Implementierung, aber die Effizienz wird verbessert.

Der Code verwendet SIMD an zwei Stellen.

1. Wenn mindestens 16 Bytes der gleichen Farbe hintereinander auftreten, werden 16 Bytes auf einmal in die Ausgabe geschrieben.
2. Wenn mindestens 16 Byte nicht wiederkehrender Farben in einer Reihe vorhanden sind, werden 16 Byte gleichzeitig gelesen und in die Ausgabe geschrieben.

Der Vorteil dieser iterativen SIMD-Implementierung besteht darin, dass sie die Vorteile des ursprünglichen Codes beibehält und gleichzeitig eine höhere Dekompressionseffizienz für Bilder mit mehreren Farbwiederholungen aufweist.

Der vollständige Code befindet sich in der Datei $bmpOpt.S$.

2.6 Implementierung in C mit SIMD-Idee

Ursprünglich wurde eine Erweiterung von SSE, d.h. Intel Intrinsics, verwendet. Nach einigen Experimenten haben wir festgestellt, dass eine Implementierung unter Verwendung der Funktionen $memset$ und $memcpy$, welche in der Bibliothek $string.h$ sind, wesentlich schnellere Laufzeiten liefert.
Deshalb werden die Intel Intrinsics SSE-Befehle in unserem Code dadurch ersetzt.

Im Allgemeinen folgt diese Implementierung noch dem Abschnitt 2.6 Implementierung in Assembler x86-64 mit SIMD, allerdings mit geänderten Implementierungsdetails.

Der vollständige Code findet sich in $bmp\_rld\_C\_opt$ in der Datei $main.c$.

3 Korrektheit

In diesem Projekt soll “Korrektheit“ ausgewählt wird, da RLE ein verlustfreier Kompressionsalgorithmus ist. Die Dekompression erfolgt nach denselben Regeln wie die Kompression und die wiederhergestellten Daten sind mit den vorkomprimierten Daten identisch, nicht nur ähnlich, so dass “Genauigkeit“ hierfür eindeutig nicht ausreicht.

Um die Korrektheit zu überprüfen, werden Bilder mit den verschiedenen Implementierungen dekomprimiert und mit dem Originalbild verglichen.

3.1 Testcases

Zunächst wurde versucht, im Internet nach Bitmaps zu suchen, die mit der RLE-Kodierung komprimiert wurden, sowie die Originalbilder. Es konnten allerdings keine Testfälle im Bitmap-Format gefunden werden, sondern ausschließlich im JPEG-Format, welches weniger Speicherplatz benötigt und für die Übertragung im Netz besser geeignet ist.

Anschließend wurden die Originalbilder mit Hilfe der Online-Kompression im Internet automatisch komprimiert, aber die Online-Kompression war nicht die verlustfreie Kompression, die für die Aufgabe erforderlich war, oder es wurde keine RLE-Kodierung verwendet.

Nach Diskussionen und Experimenten wurde schließlich Photoshop $v23.4.1$ verwendet [5], um manuell Bilder zu erstellen.

Am Anfang wird sichergestellt, dass der Typ des von Photoshop erzeugten Ergebnisses “Bitmap“ ist. Ist dies der Fall, werden die Daten im Header byteweise verglichen, da die Informationen über die Breite, Höhe und Größe des Bildes aus dem Header ausgelesen werden können. Anschließend werden diese Daten, die sogenannte Pixelanordnung, byteweise verglichen. [8]

3.2 Fehlerbeispiel

Zunächst lieferte die ursprüngliche Lösungsansatz nicht perfekte Ergebnisse. Das Bild wurde insgesamt wiederhergestellt, aber es ist offensichtlich, dass es Pixelverschiebungen im Bild gibt.

Abbildung 3: Fehlerbeispiel

Abbildung 4: Korrekte Ausgabe

(Nur repräsentative Beispiele, weitere Tests finden sich in der praktischen Abteilung.)

Die folgenden Aspekte müssen berücksichtigt werden, um die Korrektheit der Lösungsansatz zu gewährleisten.

3.3 Behandlung spezieller Marker

Es gibt Bytes in den RLE-Daten, die nicht farbbezogen sind oder die Standardfarbe darstellen. Diese Datensegmente können nicht einfach ignoriert werden, d.h. der Speicherplatz für diese Bytes sollte reserviert und vollständig durch $0x00$ ersetzt werden. Andernfalls haben die Daten einen unerwarteten Offset.

Standardmäßig werden die Pixel mit einem Wert von 0 die am häufigsten auftritten Farben im Bild darstellen. Es ist effizienter, diesen Pixeln durch den Delta-Marker, Zeilenvorschub-Marker oder End-Marker direkt den Wert 0 anzuweisen.

3.4 Behandlung von Byte Alignment

Das Bitmap-Format legt fest, dass die Anzahl der Bytes, die eine gescannte Zeile belegt, ein Vielfaches von 4 sein muss.

Andernfalls wird es mit Nullen aufgefüllt[3], d.h. diese mit Photoshop erzeugten Bilder enthalten immer 0–3 Bytes Padding am Ende der Datei, z.B.

im 8bpp-BMP-Format

- $width = 5$, jede Zeile belegt 5 Bytes, am Ende von buf wird mit 3 $0x00$ gefüllt
- $width = 6$, jede Zeile belegt 6 Bytes, am Ende von buf wird mit 2 $0x00$ gefüllt
- $width = 7$, jede Zeile belegt 7 Bytes, am Ende von buf wird mit 1 $0x00$ gefüllt

3.5 Konkrete Behandlungen

Für die Behandlung spezieller Marker werden zusätzliche Variablen $uint8\_t\ pass$ und $uint8\_t\ linecounter$ deklariert.

Nach der Verarbeitung der Pixel wird der Wert von $linecounter$ aktualisiert. Schließlich wird die Anzahl der 0-Werte am Ende jeder Zeile damit ausgerechnet.

$$pass = width - linecounter;$$

Für die Behandlung von Byte Alignment werden die zusätzliche Variable

$$uint8\_t\ linePadd = (4 - width\ \%\ 4)\ \%\ 4;$$

deklariert. $linePadd$ bestimmt die Anzahl der Paddings, die die Byte-Alignment erforderlich sind.

Insbesondere für den Delta-Marker wird die Gesamtzahl der Paddings damit ausgerechnet.

$$pass = up * width + right + linePadd * up;$$

4 Performanzanalyse

Hinweis: Nicht jede Implementierung ist effizient und manchmal kann sich die Performanz sogar verschlechtern.

4.1 Allgemeine Information

Das verwendete System hat folgende Spezifikationen: Intel(R) Core(TM) i7-10750H CPU, 2.60 GHz, 16 GB Arbeitsspeicher, Ubuntu 18.04.6 LTS, 64 bit, Linux Kernel 5.4.0-122-generic. Kompiliert wurde mit GCC 7.5.0 mit der Option -O2.

Um die Unterschiede deutlich darzustellen, wurden die Laufzeiten mit verschiedenen RLE-komprimierten Bildern berechnet. Die ausgewählten Eingabebilder sind repräsentativ und daher ausreichend, um die Performanz jeder Implementierung zu bestimmen.

1. Lena —– wird von dem Projekt zur Verfügung gestellt und ist ein klassischer Anwendungsfall im Bereich der Bildverarbeitung. “Lena“ hat relativ unregelmäßige Farben. Die Pixelwiederholungen sind gering.
2. Pinktrees —– ist ähnlich wie “Lena“, aber es ist etwa 3 Mal größer als das erste.
3. Black white —– hat relativ homogene Farben und eine große Anzahl von sich wiederholenden Pixelblöcken.
4. Black hole —– ist ähnlich wie “Black white“, aber seine Dateigröße ist viel größer als die der ersten drei, ca. 20 Mal größer, d.h. das zweite Bild hat eine höhere Auflösung.

Die Berechnungen wurden jeweils 5-mal durchgeführt. Das arithmetische Mittel jeder Eingabegröße wurde in den Grafiken eingetragen.

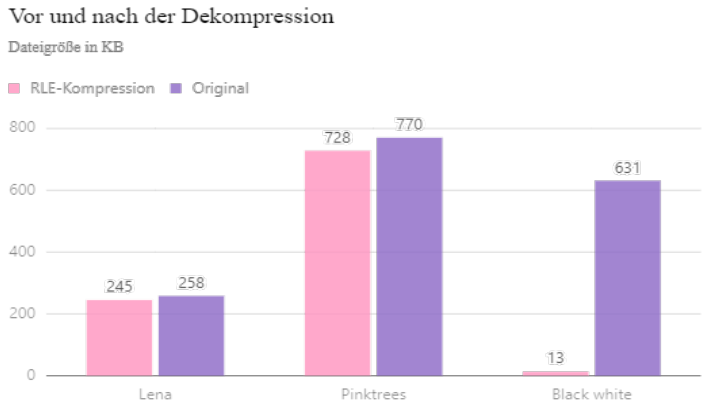
4.2 Ergebnisse

Die Analyse wird auf der Steuerungsvariable basieren, d.h. dass die Effizienz verschiedener Implementierungen mit derselben Eingabe verglichen wird.

4.2.1 Datenmenge vor und nach der Dekompression

“Black white“ hat die relativ homogenen Farben, z.B. zahlreiche schwarze oder weiße Pixel. Die wiederholenden Pixelblöcken können erheblich auf nur 2 Bytes komprimiert werden, deswegen hat es einen großen Unterschied in der Dateigröße vor und nach der Kompression.

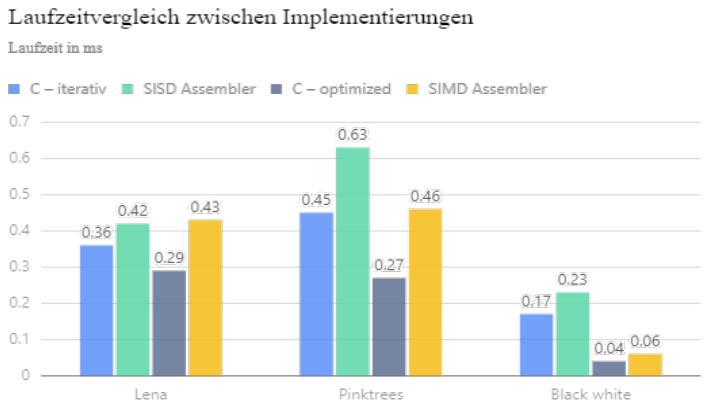
Im Gegensatz dazu haben Bilder wie “Lena“ and “Pinktrees“, die mit wenigen Farbwiederholungen sind, eine sehr geringe Kompressionsrate. Im Extremfall kann die Datenmenge durch die RLE-Kompression sogar noch größer werden.



Die getesteten Bilder befinden sich im Ordner $TestImages$, nicht im $Ausarbeitung/diagram$.

Daraus wird geschlossen, dass die Höhe der RLE-Kompressionsrate abhängig weitgehend von den Eigenschaften des Bildes selbst ist. Je größer der Bildblock mit der gleichen Farbe, desto höher ist die erzielte Kompressionsrate.

4.2.2 Laufzeitvergleich zwischen Implementierungen



Aus den Grafiken geht hervor, dass der C-Code aufgrund der Verwendung von O2 bei der Kompilierung sogar effizienter als der Assembler-Code ist. Compiler Optimierungsstufe 2 kombiniert die Optimierungen von O1 und viele andere Optimierungen, z.B.

- $-fgcse$, um Redundanzen zu vermeiden

- $-fstrength-reduce$ und $-frerun-cse-after-loop$ entfernen die von der Schleife erzeugten Variablen und zerlegen sie
- $-fcse-follow-jumps$ optimiert Sprunganweisungen, z.B. if-else-Anweisungen

Darüber hinaus ist der optimierte C-Code effizienter, insbesondere im Fall der Eingaben "Black white". Das liegt daran, dass "Black white" die meisten Farbwiederholungen aufweist.

Im Vergleich zu unserer Assembler-SIMD Implementierung ist die optimierte C Implementierung aufgrund der Verwendung von $memset$ und $memcpy$ wesentlich schneller. Wenn man den Code von $bmp\_rld\_C\_opt$ disassembliert, kann man sehen, dass diese beide Funktionen SIMD-Instruktionen zusammen mit anderen Optimierungen wie ausgerichtete Speicherzugriffe, Prefetching, "Loop unrolling" und auch CPU-spezifische Optimierungen verwenden.

Daraus lassen sich zwei Schlüsse ziehen:

1. $memset$ und $memcpy$ sind für diese Aufgabe sehr gut geeignet, da eine große Menge an wiederholenden Daten kopiert werden muss.
2. Die Laufzeit der Dekompression nach der Optimierung ist umgekehrt proportional zur Menge an wiederholenden Daten.

Dies entspricht auch der Formel für die Informationsentropie: [6]

$$H(X) = \sum_{x \in \chi} p(x) I(x) = -\sum_{x \in \chi} p(x) \log_2(p(x)).$$

Je höher die Wahrscheinlichkeit des Auftretens eines Symbols ist, desto weniger gültige Informationen repräsentiert es und desto weniger Verbrauch Dekompression benötigt.

4.2.3 Allgemeiner Laufzeitvergleich

Der allgemeine Vergleich zeigt, dass "Black hole" viel längere Zeit als "Lena" für die Dekompression braucht. Diese wird durch die Größe des Bildes, sogenannte Auflösung, bestimmt.

- Die Laufzeit des optimierten C für "Black hole" beträgt 1,02ns und für "Lena" 0,29ns.
- Die Dateigröße für "Black Hole" beträgt 14.402 KB und für "Lena" 258 KB.

Die Größe der Daten ist proportional zur Laufzeit der Dekompression. Selbst mit einem optimierten Algorithmus haben die großen Dateien aufgrund "des absoluten Unterschieds" in der Dateigröße längere Laufzeit als die kleinen Dateien.

4.3 Zusammenfassung der Performanzanalyse

Aus der Performanzanalyse lässt sich zusammenfassen, dass der Compiler mit einer Optimierungsstufe von -O2 den Code der Implementierung in C merklich optimiert. Seine Performanz übertrifft sogar handgeschriebenen x86-64 Assemblercode. Die Parallelisierung von Algorithmen über SIMD bietet die beste Performanz. Dies gilt sowohl für Implementierung in Assembler x86-64 mit SIMD als auch für optimierte C mit $memset$ und $memcpy$. Aufgrund der hohen Effizienz von $memset$ und $memcpy$ erzielt die optimierte C-Implementierung die besten Ergebnisse.

5 Zusammenfassung und Ausblick

5.1 Ausblick

Wie bei allen Projekten gibt es immer Raum für Verbesserungen.

Eine mögliche Optimierung ist, dass der Aufwand der Schleife aufgrund der geringen Anzahl von Datenwiederholungen größer als bei der direkten Ausführung ist. Wenn die Anzahl der Pixelwiederholungen klein ist, z.B. weniger als 3 mal, kann man die Daten direkt kopieren.

Eine weitere mögliche Optimierung ist, dass eine externe Maske vor dem Lese- und Färbezyklus auf null gesetzt wird. Während jedes Pixelvergleichs wird das Ergebnis in der internen Maske gespeichert und die externe Maske wird dann aktualisiert. Die interne Maske zeigt an, welche der 16 Pixel erfolgreich oder nicht erfolgreich waren und daher gefärbt oder übersprungen werden sollten (oder die Default-Farbe).

Dies passt sich auch an Lese- und Schreibvorgänge mit Alignment an und wird die Effizienz von SIMD weiter verbessern.

5.2 Zusammenfassung

Durch dieses Praktikum hat unsere Gruppe ein erstes Verständnis für die Bilddekompression, insbesondere der Dekodierung von RLE-Daten.

Im Großen und Ganzen wird aus den unterschiedlichen Implementierungen im Allgemeinen deutlich, dass es wichtig ist, die Zielarchitektur des Projekts vollständig zu verstehen und sie möglichst optimal zu verwenden, um die Performanz zu verbessern. Obwohl der GCC-Compiler den Code optimiert, ist die korrekte Beherrschung und Verwendung von SIMD definitiv ein notwendiger Optimierungsansatz.

Literatur

[1] Bassiouni. *Data Compression in Scientific and Statistical Databases*. IEEE Trans. Software Eng., October 1985.

[2] Claude E. Shannon. *A mathematical theory of communication*. Bell Labs Technical Journal, October 1948. `https://dl.acm.org/doi/pdf/10.1145/584091.584093`, visited 2022-07-19.

[3] daleg38832663. *BMP file has extra padding. Is this byte alignment?* Adobe Support Community, May 2020. `https://community.adobe.com/t5/photoshop-ecosystem-discussions/bmp-file-has-extra-padding-is-this-byte-alignment/td-p/11159234`, visited 2022-07-16.

[4] Dwayne Phillips. *Image Processing in C*. R & D Publications, 1994. `https://pdfslide.net/documents/phillips-dimage-processing-with-c.html?page=1`, visited 2022-07-04.

[5] fileinfo. *.BMP File Extension*. fileinfo, April 2022. `https://fileinfo.com/extension/bmp`, visited 2022-07-14.

[6] Lehrstuhl für Netzarchitekturen und Netzdienste. *Grundlagen Rechnernetze und Verteilte Systeme (GRNVS)*. Technische Universität München, April 2022. `https://grnvs.net.in.tum.de/slides_chap1.pdf`, visited 2022-07-17.

[7] Microsoft. *Bitmap Compression*. Microsoft, August 2021. `https://docs.microsoft.com/en-us/windows/win32/gdi/bitmap-compression?redirectedfrom=MSDN`, visited 2022-06-29.

[8] Microsoft. *Bitmap Storage*. Microsoft, January 2021. `https://docs.microsoft.com/en-us/windows/win32/gdi/bitmap-storage`, visited 2022-07-15.

[9] Prof. Dr. Elke Hergenröther. *Graphische Datenverarbeitung*. Hochschule Darmstadt, October 2019. `https://fbi.h-da.de/fileadmin/Personen/fbi1111/GDV/vorlesung/SS1920_7_GDV_Kompression.pdf`, visited 2022-06-29.